Eclipse's Rich Client Platform, Part 2: Extending the generic workbench

Skill Level: Intermediate

Jeff Gunther (jeff.gunther@intalgent.com)
General Manager
Intalgent Technologies

27 Jul 2004

This tutorial, the second in the "Eclipse's Rich Client Platform" two-part series, continues exploring the Eclipse Rich Client Platform by expanding the previous discussion. It demonstrates how to use views, actions, and wizards to assemble a complete application.

Section 1. Before you start

About this tutorial

The second of a two-part series, this tutorial explores Eclipse's Rich Client Platform (RCP). Part 1 began with a review of the Eclipse project and the relevance of the RCP within the marketplace. It discusses the Eclipse plug-in architecture and outlines the necessary steps to implement an RCP application. After providing the necessary background information, you began creating a project within the Eclipse V3.0 IDE. You defined a plug-in manifest, were introduced to extensions and extension points, and created a basic perspective. Using these components, you created some additional supporting Java™ classes and launched a stand-alone RCP application.

Part 2 leverages the discussion from Part 1 and explores how to use other Eclipse user-interface (UI) components, such as views, actions, and wizards, to assemble a complete application. In this tutorial, you'll create a front end for the Google API that will give you the ability to query and display search results from Google's extensive catalog of Web sites. Having an application that demonstrates some of these technologies in action will provide you with an understanding of the RCP platform.

Prerequisites

You should understand how to navigate Eclipse V3.0 and have a working knowledge of Java technology to follow along. You do not need a background in Eclipse plug-in development or an understanding of technologies such as the Standard Widget Toolkit (SWT) and JFace. Part 1 provides a brief introduction to each of these complementary technologies. This tutorial explores the code and supporting files so you can grasp how to construct an RCP application.

System requirements

While not a requirement, you'll find this tutorial easier to follow if you download, install, and configure Eclipse V3.0, a 1.4 Java Virtual Machine, and Apache Ant. If you don't have these tools installed, please reference, download, and install the following resources:

- Eclipse V3.0
- Java 2 Standard Edition, Software Development Kit (SDK)
- Apache Ant V1.6.1

Section 2. Defining a view

Overview of views

Views within the Eclipse workbench are visual containers that allow users to display or navigate resources of a particular type. As you begin creating views within your own RCP application, remember to review the view's purpose before starting development. Since a view's responsibility is to display data from your domain model, group similar types of objects into the view. For example, most users of the Eclipse IDE make extensive use of the tasks view within the Java perspective. This view displays types of auto-generated errors, warnings, or information associated with a resource you need to review and resolve. This approach minimizes the need for the user to toggle between views to accomplish a particular task. The number of views any application has is largely dependent on the application's size and complexity. The example Google application developed in this tutorial has two views -- one for searching and one for displaying the Web page from the search results.

Defining an org.eclipse.ui.views extension

Similar to other components within Eclipse, to create a new view, you must define a new extension within the project's plug-in manifest. You define views using the `org.eclipse.ui.perspectives` extension point. Using the plugin.xml tab of the plug-in manifest editor within the Google project, add the following content to begin the process of creating the views.

Listing 1. Use plugin.xml tab of plug-in manifest editor within Google project

```
...
      <extension point="org.eclipse.ui.views">

                  <category
                        id="com.ibm.developerworks.google.views"
                        name="Google">
                  </category>

                  <view
                        id="com.ibm.developerworks.google.views.SearchView"
                        name="Search"
                        category="com.ibm.developerworks.google.views"
                        class="com.ibm.developerworks.google.views.SearchView"
                        icon="icons/google.gif">
                  </view>

                  <view
                        id="com.ibm.developerworks.google.views.BrowserView"
                        name="Browser"
                        category="com.ibm.developerworks.google.views"
                        class="com.ibm.developerworks.google.views.BrowserView"
                        icon="icons/google.gif">
                  </view>

      </extension>
...
```

The `SearchView` allows users to search Google and display the search results in a table. The `BrowserView` contains an SWT browser control and displays a particular URL based on the user's action within the search results table.

Stepping through the org.eclipse.ui.views extension point

Use the `<extension>`, `<category>`, and `<view>` elements to define the view extension point. A category is used within the Show View Dialog to group similar views. Each view can appear under multiple categories.

The `<category>` element has the following attributes:

- `id` -- This defines a unique identifier for the category.
- `name` -- This defines a name for this category, and the workbench uses it to represent this category.
- `parentCategory` -- This optional attribute defines a list of categories separated by '/'. This element creates category hierarchies.

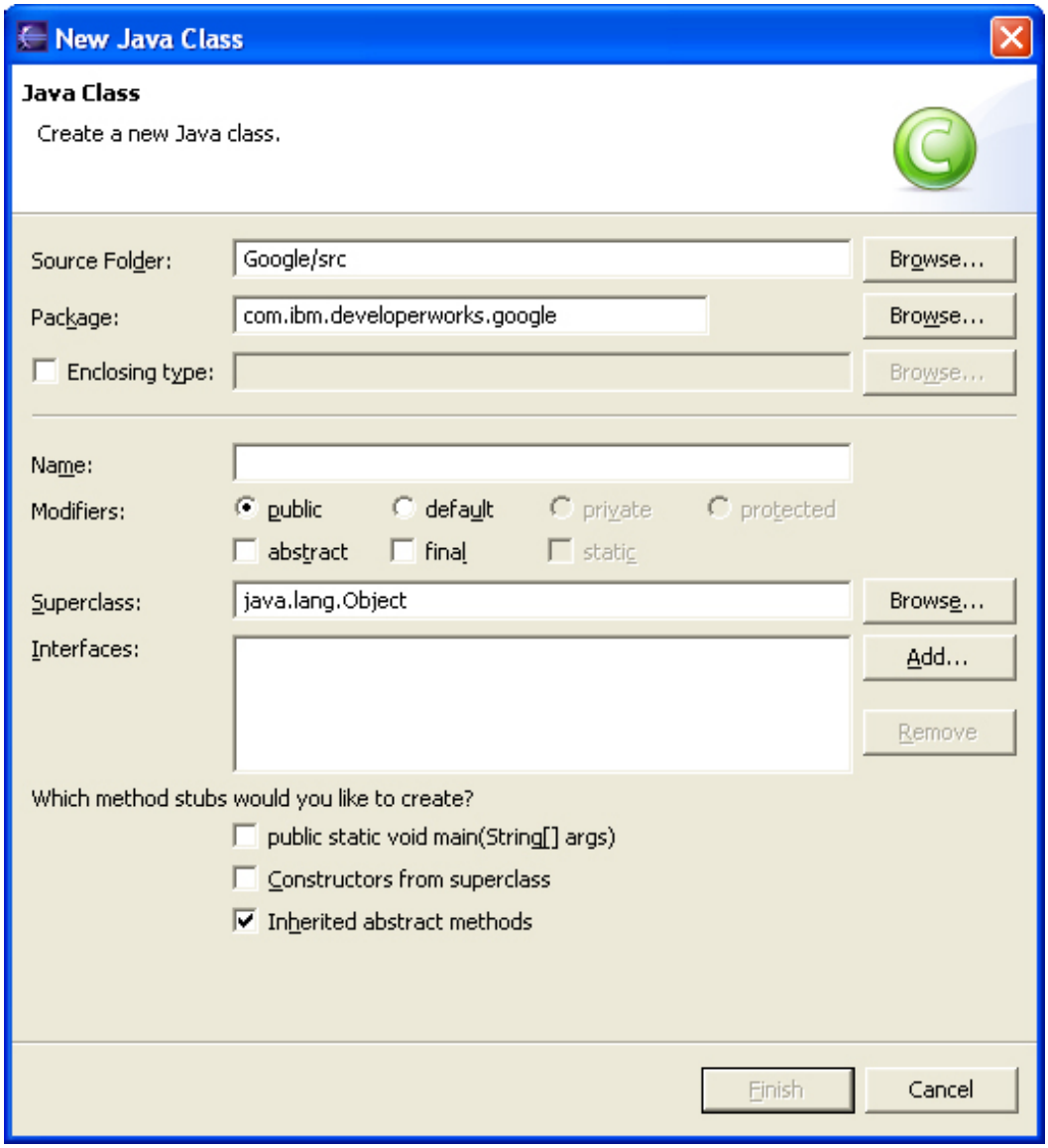
The `<view>` element has the following attributes:

- `id` -- This defines a unique identifier for the view.
- `name` -- This defines a name for this view, and the workbench uses it to represent this view.
- `category` -- This optional attribute defines the categories identifiers. Each category is separated by a '/' and must exist within the plug-in manifest prior to being referenced by the `<view>` element.
- `class` -- This contains the fully-qualified name of the class that implements the `org.eclipse.ui.IViewPart` interface.
- `icon` -- This optional attribute contains a relative name of the icon associated with the view.
- `fastViewWidthRatio` -- This optional attribute contains the percentage of the width of the workbench the view will take up. This attribute must be a floating point value between 0.05 and 0.95.
- `allowMultiple` -- This optional attribute indicates whether this view allows for the creation of multiple instances within the workbench.

Creating the SearchView class

To create the `SearchView` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
   Figure 1. New Java Class wizard



2. Type `com.ibm.developerworks.google.views` in the Package field.
3. Type `SearchView` in the Name field.
4. Click **Browse** to display the Superclass Selection dialog box.
5. Type `org.eclipse.ui.part.ViewPart` in the Choose a Type field and click **OK**.
6. Click **Finish** to create the new class.

Implementing the SearchView class

After the class is created, the `createPartControl` and `setFocus` methods must be implemented. The `createPartControl` method is responsible for creating the

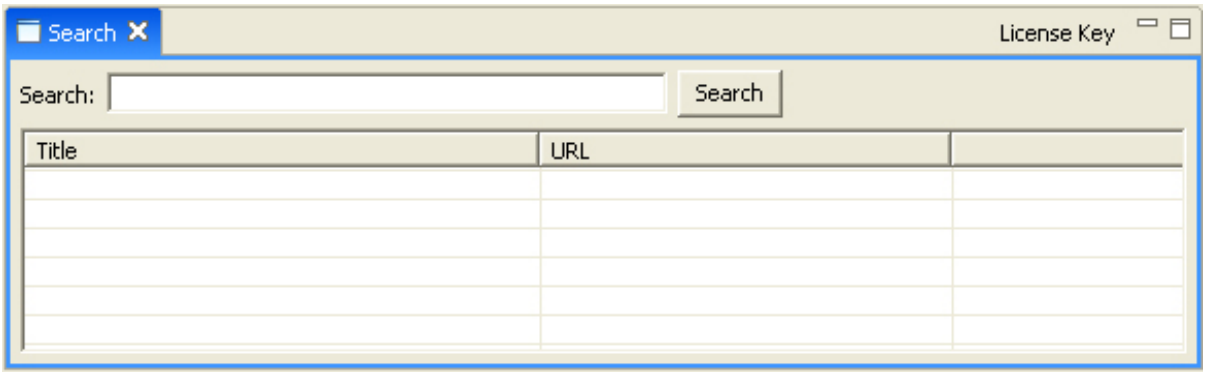
view's UI controls. In this case, an SWT GridLayout layout is used to arrange the SWT label, SWT text, SWT button, and an SWT table on the view's composite. For more information about SWT's various layouts or how to use SWT UI components, please refer to Resources.

The code within the `createPartControl` method renders the UI Figure 2 shows.

Listing 2. Code within createPartControl

```
...
    public void createPartControl(Composite parent)
    {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 3;
        gridLayout.marginHeight = 5;
        gridLayout.marginWidth = 5;

        parent.setLayout(gridLayout);

        Label searchLabel = new Label(parent, SWT.NONE);
        searchLabel.setText("Search:");

        searchText = new Text(parent, SWT.BORDER);
        searchText.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
                | GridData.HORIZONTAL_ALIGN_FILL));

        Button searchButton = new Button(parent, SWT.PUSH);
        searchButton.setText(" Search ");
...
        GridData gridData = new GridData();
        gridData.verticalAlignment = GridData.FILL;
        gridData.horizontalSpan = 3;
        gridData.grabExcessHorizontalSpace = true;
        gridData.grabExcessVerticalSpace = true;
        gridData.horizontalAlignment = GridData.FILL;

        tableViewer = new TableViewer(parent, SWT.FULL_SELECTION | SWT.BORDER);
        tableViewer.setLabelProvider(new SearchViewLabelProvider());
        tableViewer.setContentProvider(new ViewContentProvider());
        tableViewer.setInput(model);
        tableViewer.getControl().setLayoutData(gridData);
        tableViewer.addDoubleClickListener(this);

        Table table = tableViewer.getTable();
        table.setHeaderVisible(true);
        table.setLinesVisible(true);

        TableColumn titleColumn = new TableColumn(table, SWT.NONE);
        titleColumn.setText("Title");
        titleColumn.setWidth(250);

        TableColumn urlColumn = new TableColumn(table, SWT.NONE);
        urlColumn.setText("URL");
        urlColumn.setWidth(200);
...
```

Figure 2. Search view of the Google application



In addition to the `createPartControl` method, the `setFocus` method must be implemented. In this case, the focus defaults to an SWT Text field that allows a user to input search criteria for Google. This method is called upon the view being rendered within the workbench.

Listing 3. Method called upon view being rendered

```
...
    public void setFocus()
    {
        searchText.setFocus();
    }
...
```

Once a user double-clicks on a row within the search results table, the Web site loads within another view that contains an SWT browser control. This is accomplished by having the `SearchView` implement the `IDoubleClickListener` interface. The `IDoubleClickListener` interface requires a `doubleClick` method to be added to the `SearchView`.

Listing 4. IDoubleClickListener interface requires doubleClick method to be added to SearchView

```
...
    public void doubleClick(DoubleClickEvent event)
    {
        if (!tableViewer.getSelection().isEmpty())
        {

            IStructuredSelection ss = (IStructuredSelection) tableViewer
                    .getSelection();
            GoogleSearchResultElement element = (GoogleSearchResultElement) ss
                    .getFirstElement();

            BrowserView.browser.setUrl(element.getURL());
        }

    }
...
```

Find the complete source code for the `SearchView` class below:

Listing 5. SearchView class source code

```
package com.ibm.developerworks.google.views;

import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.DoubleClickEvent;
import org.eclipse.jface.viewers.IDoubleClickListener;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Table;
import org.eclipse.swt.widgets.TableColumn;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.internal.dialogs.ViewContentProvider;
import org.eclipse.ui.part.ViewPart;

import com.google.soap.search.GoogleSearch;
import com.google.soap.search.GoogleSearchFault;
import com.google.soap.search.GoogleSearchResult;
import com.google.       soap.search.GoogleSearchResultElement;
import com.ibm.developerworks.google.GoogleApplication;

public class SearchView extends ViewPart implements IDoubleClickListener
{
    public static final String ID = "com.ibm.developerworks.google.views.SearchView";

    private TableViewer tableViewer;

    private Text searchText;

    private GoogleSearchResultElement model;

    public void createPartControl(Composite parent)
    {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 3;
        gridLayout.marginHeight = 5;
        gridLayout.marginWidth = 5;

        parent.setLayout(gridLayout);

        Label searchLabel = new Label(parent, SWT.NONE);
        searchLabel.setText("Search:");

        searchText = new Text(parent, SWT.BORDER);
        searchText.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
                | GridData.HORIZONTAL_ALIGN_FILL));

        Button searchButton = new Button(parent, SWT.PUSH);
        searchButton.setText(" Search ");
        searchButton.addSelectionListener(new SelectionListener()
        {

            public void widgetSelected(SelectionEvent e)
            {

                GoogleSearch search = new GoogleSearch();
                search.setKey(GoogleApplication.LICENSE_KEY);
                search.setQueryString(searchText.getText());
                try
                {
                    GoogleSearchResult result = search.doSearch();

                    tableViewer.setInput(model);
                    tableViewer.add(result.getResultElements());

                } catch (GoogleSearchFault ex)
                {
                    MessageDialog.openWarning(e.display.getActiveShell(),
```

```
                            "Google Error", ex.getMessage());
                }

            }

            public void widgetDefaultSelected(SelectionEvent e)

            }
        });

        GridData gridData = new GridData();
        gridData.verticalAlignment = GridData.FILL;
        gridData.horizontalSpan = 3;
        gridData.grabExcessHorizontalSpace = true;
        gridData.grabExcessVerticalSpace = true;
        gridData.horizontalAlignment = GridData.FILL;

        tableViewer = new TableViewer(parent, SWT.FULL_SELECTION | SWT.BORDER);
        tableViewer.setLabelProvider(new SearchViewLabelProvider());
        tableViewer.setContentProvider(new ViewContentProvider());
        tableViewer.setInput(model);
        tableViewer.getControl().setLayoutData(gridData);
        tableViewer.addDoubleClickListener(this);

        Table table = tableViewer.getTable();
        table.setHeaderVisible(true);
        table.setLinesVisible(true);

        TableColumn titleColumn = new TableColumn(table, SWT.NONE);
        titleColumn.setText("Title");
        titleColumn.setWidth(250);

        TableColumn urlColumn = new TableColumn(table, SWT.NONE);
        urlColumn.setText("URL");
        urlColumn.setWidth(200);

    }

    public void setFocus()
    {
        searchText.setFocus();
    }

    public void doubleClick(DoubleClickEvent event)
    {
        if (!tableViewer.getSelection().isEmpty())
        {

            IStructuredSelection ss = (IStructuredSelection) tableViewer
                    .getSelection();
            GoogleSearchResultElement element = (GoogleSearchResultElement) ss
                    .getFirstElement();

            BrowserView.browser.setUrl(element.getURL());
        }

    }
}
```

Creating the SearchViewLabelProvider class

In the code in the previous section, the `TableViewer` object uses a class called `SearchViewLabelProvider`. In this instance, a label provider sets the column's text for each row of the table. To create the `SearchViewLabelProvider` class for the `SearchView` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java

Class wizard.

2. Type `com.ibm.developerworks.google.views` in the Package field.
3. Type `SearchViewLabelProvider` in the Name field.
4. Click **Browse** to display the Superclass Selection dialog box.
5. Type `org.eclipse.jface.viewers.LabelProvider` in the Choose a Type field and click **OK**.
6. Click **Add** to display the Implemented Interfaces Selection dialog box.
7. Type `org.eclipse.jface.viewers.ITableLabelProvider` in the Choose an interface field and click **OK**.
8. Click **Finish** to create the new class.

Implementing the SearchViewLabelProvider class

The `ITableLabelProvider` interface requires that the `getColumnImage` and `getColumnText` methods be implemented within the class. Since the results table does not include any images, the `getColumnImage` method simply returns null. The `getColumnText` uses the `GoogleSearchResultElement` class provided by the Google API to set the first and second columns of the SWT table. The first column contains the title of the search result, and the second column contains the search result's URL.

Listing 6. getColumnText uses GoogleSearchResultElement class provided by Google API

```
package com.ibm.developerworks.google.views;

...

public class SearchViewLabelProvider extends LabelProvider implements
        ITableLabelProvider
{

    public Image getColumnImage(Object element, int columnIndex)
    {
        return null;
    }

    public String getColumnText(Object element, int columnIndex)
    {
        switch (columnIndex)
        {
        case 0:
            return ((GoogleSearchResultElement) element).getTitle();
        case 1:
            return ((GoogleSearchResultElement) element).getURL();

        }
        return "";
```

```
    }
}
```

Creating the BrowserView class

Now you need to create a view to display the URL the user selects within the search result table. To create the `BrowserView` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type `com.ibm.developerworks.google.views` in the Package field.
3. Type `BrowserView` in the Name field.
4. Click **Browse** to display the Superclass Selection dialog box.
5. Type `org.eclipse.ui.part.ViewPart` in the Choose a Type field and click **OK**.
6. Click **Finish** to create the new class.

Implementing the BrowserView class

As for the `SearchView` class, you must implement the `createPartControl` and `setFocus` methods in the `BrowserView` class. In this case, an SWT browser control is embedded within the view. This control displays the Web page that the user selects within the search results table.

Listing 7. SWT browser control embedded within view

```
package com.ibm.developerworks.google.views;

import org.eclipse.swt.SWT;
import org.eclipse.swt.browser.Browser;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class BrowserView extends ViewPart
{
    public static final String ID = "com.ibm.developerworks.google.views.BrowserView";

    public static Browser browser;

    public void createPartControl(Composite parent)
    {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 1;
```

```
        gridLayout.marginHeight = 5;
        gridLayout.marginWidth = 5;
        parent.setLayout(gridLayout);

        browser = new Browser(parent, SWT.NONE);

        browser.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
                | GridData.GRAB_VERTICAL | GridData.FILL_HORIZONTAL
                | GridData.FILL_VERTICAL));
        browser.setUrl("about:");

    }

    public void setFocus()
    {
        browser.setFocus();

    }
}
```

Integrating the SearchView and BrowserView into a perspective

With the two views and supporting classes defined for your Google application, you need to integrate these components into the existing perspective you created in Part 1. Open the `GooglePerspective` class and modify the `createInitialLayout` method.

Find the complete code for the `GooglePerspective` class below:

Listing 8. GooglePerspective class

```
package com.ibm.developerworks.google;

import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

import com.ibm.developerworks.google.views.BrowserView;
import com.ibm.developerworks.google.views.SearchView;

public class GooglePerspective implements IPerspectiveFactory
{
    public static final String ID = "com.ibm.developerworks.google.GooglePerspective";

    public void createInitialLayout(IPageLayout layout)
    {
        layout.setEditorAreaVisible(false);
        layout.addView(SearchView.ID, IPageLayout.BOTTOM, new Float(0.60)
                .floatValue(), IPageLayout.ID_EDITOR_AREA);
        layout.addView(BrowserView.ID, IPageLayout.TOP, new Float(0.40)
                .floatValue(), IPageLayout.ID_EDITOR_AREA);
    }
}
```

As Figure 3 shows, the last two lines within the `createInitialLayout` open the `SearchView` and `BrowserView` when the perspective is rendered. The `addView` method contains four parameters:

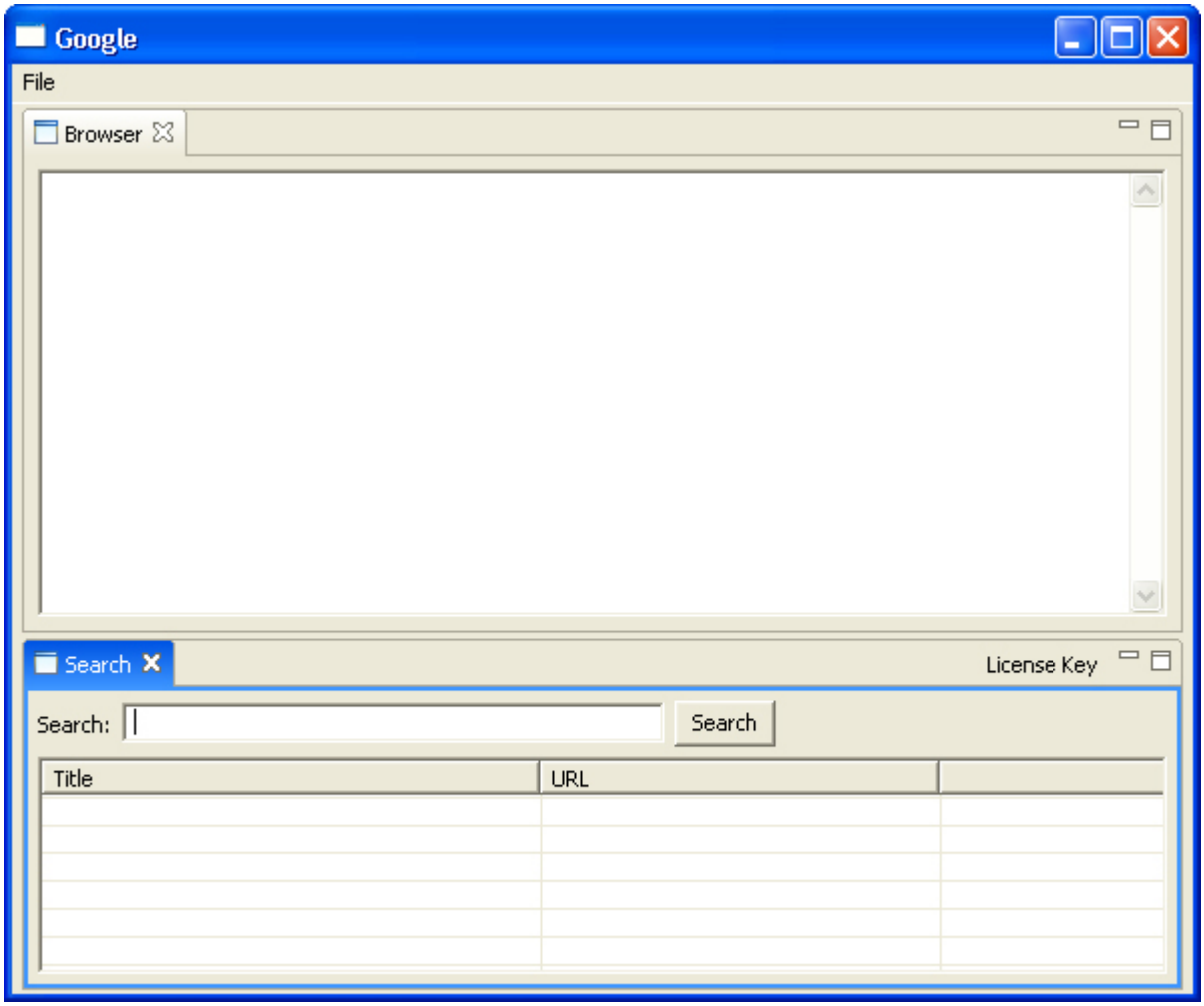
1. The first contains the unique identifier for the view.

2. The second defines the relationship to the workbench. Possible options include Top, Bottom, Left, and Right.

3. The third defines a ratio of how to divide the available space within the workbench. This parameter is a float value between 0.05 and 0.95

4. The final parameter contains the unique identifier reference for where the view should be displayed. In this case, the editor area is used.

Figure 3. The Search and Browser views of the Google application



The next section focuses on how to add menu bars, dialogs, action, and wizards to an RCP application.

Section 3. Integrating menu bars and dialog boxes

Adding menu bars to a perspective

Sometimes you'll want to add actions to an RCP application by creating a menu bar within the main window.

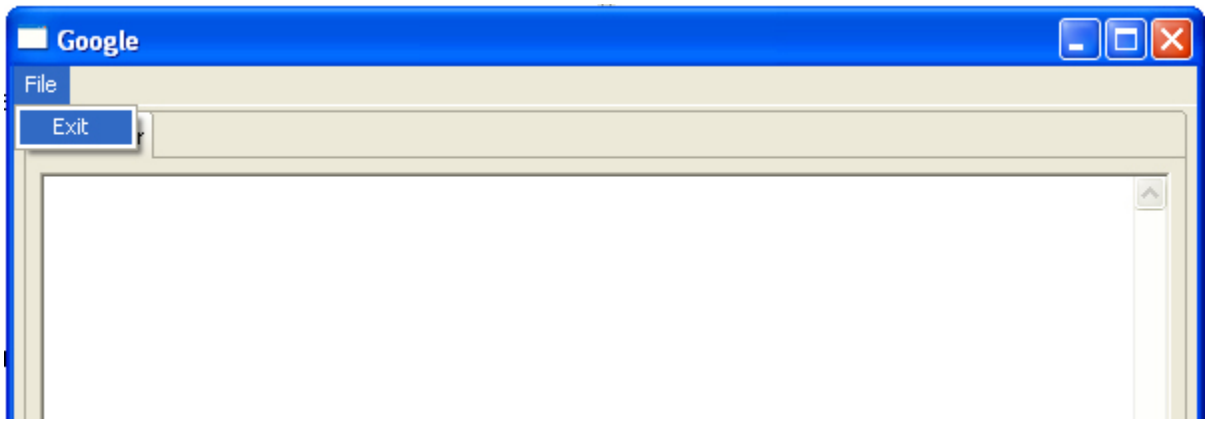
To add new items to the menu bar, you need to override the `fillActionBars` method within the `WorkbenchAdvisor`.

Listing 9. Override fillActionBars method within WorkbenchAdvisor

```
...
    public void
fillActionBars(IWorkbenchWindow
window,
            IActionBarConfigurer
configurer, int flags)
    {
        IMenuManager menuBar =
configurer.getMenuManager();

        MenuManager fileMenu = new
MenuManager("File",
IWorkbenchActionConstants.M_FILE);
        fileMenu.add(new
GroupMarker(IWorkbenchAction\
        Constants.FILE_START));
        fileMenu.add(new
GroupMarker(IWorkbenchAction\
        Constants.MB_ADDITIONS));
fileMenu.add(ActionFactory.QUIT.create(window));
        fileMenu.add(new
GroupMarker(IWorkbenchAction\
        Constants.FILE_END));

        menuBar.add(fileMenu);
    }
...
```

Above, the `MenuManager` class adds a `fileMenu` to the workbench. Figure 4 shows the menu bar in action within the Google application.

Figure 4. Menu bars in the WorkbenchAdvisor class

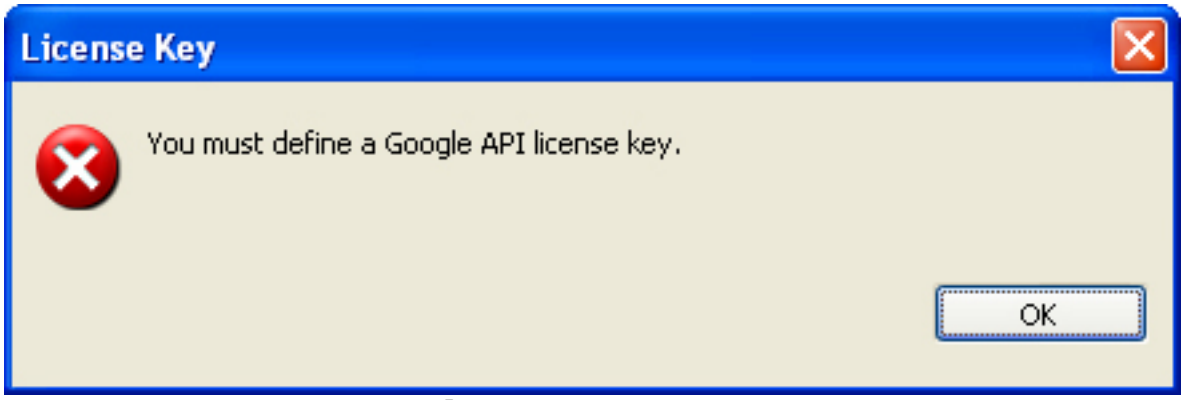


In addition to menu bars, the JFace toolkit provides some predefined dialog boxes that can enhance a user's experience with an RCP application. The next section reviews the various dialog types the JFace package provides.

Various types of dialog boxes

The JFace toolkit includes a variety of dialogs that can display messages to your application's users. As Figure 5 demonstrates, the `SearchView` class uses the `MessageDialog` class to display an error to users if they don't provide a Google API license key before executing a query.

Figure 5. Error message if the license key is not provided



In addition to the `MessageDialog`, the JFace package provides three other dialog types:

1. `ErrorDialog` uses an IStatus object and displays information about a particular error.
2. `InputDialog` allows the user to enter text into the dialog box.
3. `ProgressMonitorDialog` shows the execution progress of a process to the user.

The next section describes how you can add a wizard to your RCP application to gather data.

Section 4. Defining a wizard

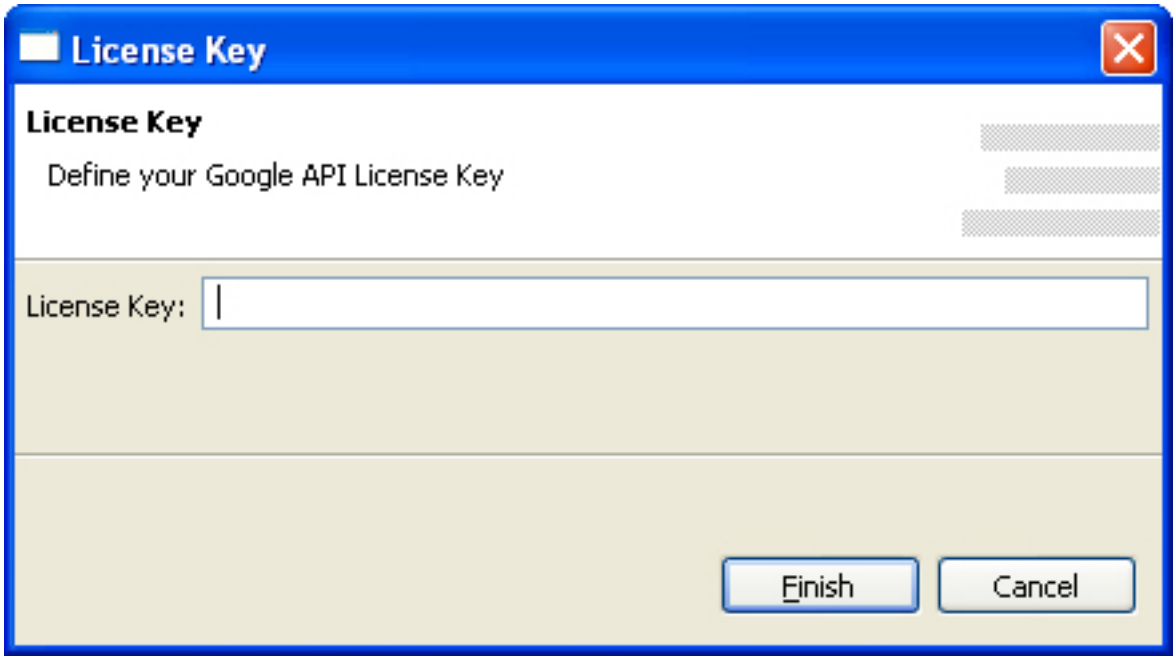
Overview of wizards

The JFace toolkit includes a powerful set of user-interface components that you can easily integrate into an RCP application. An interesting component of this toolkit is the support for wizards. A JFace wizard, coupled with other user-interface components within the Standard Widget Toolkit (SWT), provides a flexible mechanism to systematically gather user input and perform input validation.

Before reviewing the code and implementation details of how to create a wizard, review the purpose for a wizard within your Google application. To query the Google API, you must sign up for an account. Once your account has been activated, you'll be provided a license key. Google currently allows each account the ability to execute 1000 queries per day. Since you need to supply a license key as a parameter within the `GoogleSearch` object, you need a mechanism to gather the license key from the user.

As Figure 6 demonstrates, the application contains a JFace wizard consisting of one page that requests the license key.

Figure 6. The License Key wizard within the Google application



For more information on how to create an account to access the Google API, please refer to Resources.

Creating a LicenseKeyWizard class

To create a basic wizard, create a class that extends `org.eclipse.jface.wizard.Wizard`. In the Google application, a wizard will be used to gather the user's Google API license key. To create the `LicenseKeyWizard` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type `com.ibm.developerworks.google.wizards` in the Package field.
3. Type `LicenseKeyWizard` in the Name field.

4. Click **Browse** to display the Superclass Selection dialog box.
5. Type `org.eclipse.jface.wizard.Wizard` in the Choose a Type field and click **OK**.
6. Click **Finish** to create the new class.

Implementing the LicenseKeyWizard class

After creating the `LicenseKeyWizard` class, you need to override the `addPages` and `performFinish` methods. The `addPages` method adds pages to a wizard before it displays to the end user. The `performFinish` method executes when the user presses the **Finish** button within the wizard. The `LicenseKeyWizard` gathers the license key data and populates it to a static String variable in the class.

Find the complete source code for the `LicenseKeyWizard` class below:

Listing 10. LicenseKeyWizard class

```
package com.ibm.developerworks.google.wizards;

import org.eclipse.jface.wizard.Wizard;


public class LicenseKeyWizard extends Wizard
{
    private static String licenseKey;
    private LicenseKeyWizardPage page;

    public LicenseKeyWizard()
    {
        super();
        this.setWindowTitle("License Key");
    }

    public void addPages()
    {
        page = new LicenseKeyWizardPage("licenseKey");
        addPage(page);
    }
    public boolean performFinish()
    {
        if(page.getLicenseKeyText().getText().equalsIgnoreCase(""))
        {
            page.setErrorMessage("You must provide a license key.");
            page.setPageComplete(false);
            return false;
        }
        else
        {
            licenseKey = page.getLicenseKeyText().getText();
            return true;
        }

    }

    public static String getLicenseKey()
    {
        return licenseKey;
    }

    public static void setLicenseKey(String licenseKey)
```

```
    {
        LicenseKeyWizard.licenseKey = licenseKey;
    }
}
```

Creating a LicenseKeyWizardPage class

In addition to the wizard class, each wizard must have at least one page that extends `org.eclipse.jface.wizard.WizardPage`. To create the `LicenseKeyWizardPage` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type `com.ibm.developerworks.google.wizards` in the Package field.
3. Type `LicenseKeyWizardPage` in the Name field.
4. Click **Browse** to display the Superclass Selection dialog box.
5. Type `org.eclipse.jface.wizard.WizardPage` in the Choose a Type field and click **OK**.
6. Click **Finish** to create the new class.

Implementing the LicenseKeyWizardPage class

Without a class that implements a `WizardPage`, the `LicenseKeyWizard` wouldn't have any behavior. You can think of a wizard as a stack of cards, each with its own layout and design. Each `WizardPage` is responsible for the layout and behavior of a single page or card within a wizard. To create a `WizardPage`, you need to subclass the `WizardPage` base implementation and implement the `createControl` method to create the specific user-interface controls.

Find the complete source code for the `LicenseKeyWizardPage` class below:

Listing 11. LicenseKeyWizardPage class

```
package com.ibm.developerworks.google.wizards;

import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

public class LicenseKeyWizardPage extends WizardPage
```

```
{
    private Text licenseKeyText;

    protected LicenseKeyWizardPage(String pageName)
    {
        super(pageName);
        setTitle("License Key");
        setDescription("Define your Google API License Key");
    }

    public void createControl(Composite parent)
    {
        GridLayout pageLayout = new GridLayout();
        pageLayout.numColumns = 2;
        parent.setLayout(pageLayout);
        parent.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

        Label label = new Label(parent, SWT.NONE);
        label.setText("License Key:");

        licenseKeyText = new Text(parent, SWT.BORDER);
        licenseKeyText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

        setControl(parent);
    }

    public Text getLicenseKeyText()
    {
        return licenseKeyText;
    }

    public void setLicenseKeyText(Text licenseKeyText)
    {
        this.licenseKeyText = licenseKeyText;
    }
}
```

Section 5. Defining an action

Overview of actions

Actions within the Eclipse workbench are commands that the user of an application triggers. In general, actions fall into three distinct categories: buttons, items within the tool bar, or items within the menu bar. For example, when you select **File > New > Class** from the menu bar, you're executing an action that opens the New Java Class wizard. When you execute an action within the workbench, the action's run method performs its particular function within the application. In addition to an action's class, an action can have other properties that control how the action is rendered within the workbench. These properties include a text label, mouse over tool tip, and an icon. This tutorial's example Google application has two actions -- one that's used to exit the application and one that allows users to provide their Google API license key by clicking a button located on the Search view.

This section explores how to define actions with an extension point within the plug-in manifest. Specifically, it covers how to add an action to the pull-down menu of the Search view.

Defining the org.eclipse.ui.viewActions extension point

To add a new action to a view, you must define a new extension within the project's plug-in manifest. View actions are defined using the `org.eclipse.ui.viewActions` extension point. Each view has a pull-down menu that activates when you click on the top right triangle button. Using the plugin.xml tab of the plug-in manifest editor within the Google project, add the following content to begin the process of creating a view action:

Listing 12. Create view action

```
...
      <extension point="org.eclipse.ui.viewActions">

            <viewContribution
                  id="com.ibm.developerworks.google.views.contribution"
                  targetID="com.ibm.developerworks.google.views.SearchView">

                        <action
                            id="com.ibm.developerworks.google.  \
                            actions.LicenseKeyAction"
                            label="License Key"
                            toolbarPath="additions"
                            style="push"
                            state="false"
                            tooltip="Google License Key"
                 class="com.ibm.developerworks.google.actions.LicenseKeyAction" />

             </viewContribution>

      </extension>
...
```

The `LicenseKey` view action allows users to set the license key that will be used to query Google's API. The next few sections describe the `org.eclipse.ui.viewActions` extension point and the steps necessary to create an `Action class`.

Stepping through the org.eclipse.ui.viewActions extension point

Beginning with the `<extension>` element, a simple `org.eclipse.ui.viewActions` extension contains a `<viewContribution>` and `<action>` element.

A `<viewContribution>` defines a group of view actions and menus. This element has the following attributes:

1. `id` -- This defines a unique identifier for the view contribution.
2. `targetID` -- This defines a registered view that is the target of the contribution.

The `<action>` element has the following attributes:

- `id` -- This defines a unique identifier for the action.
- `label` -- This defines a name for this action and, the workbench uses it to represent this action.
- `menubarPath` -- This optional attribute contains a slash-delimited path ('/') used to specify the location of this action in the pull-down menu.
- `toolbarPath` -- This optional attribute contains a named group within the toolbar of the target view. If this attribute is omitted, the action will not appear in the view's toolbar.
- `icon` -- This optional attribute contains the relative path of an icon used to visually represent the action within the view.
- `disableIcon` -- This optional attribute contains the relative path of an icon used to visually represent the action when it's disabled.
- `hoverIcon` -- This optional attribute contains the relative path of an icon used to visually represent the action when the mouse pointer is hovering over the action.
- `tooltip` -- This optional attribute defines the text for the action's tool tip.
- `helpContextId` -- This optional attribute defines a unique identifier indicating the action's help context.
- `style` -- This optional attribute defines the user-interface style type for the action. Options include push, radio, or toggle.
- `state` -- When the style attribute is toggled, this optional attribute defines the initial state of the action.
- `class` -- This attribute contains the fully qualified name of the class that implements the `org.eclipse.ui.IViewActionDelegate` interface.

Creating the LicenseKeyAction class

To create the `LicenseKeyAction` class for the `SearchView` class, complete the following steps within the Google project:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type `com.ibm.developerworks.google.actions` in the Package field.
3. Type `LicenseKeyAction` in the Name field.
4. Click **Add** to display the Implemented Interfaces Selection dialog box.
5. Type `org.eclipse.ui.IViewActionDelegate` in the Choose an

interface field and click **OK**.

6. Click **Finish** to create the new class.

Implementing the LicenseKeyAction class

When an action is invoked, the action's run method executes. In the Google application, the `LicenseKeyAction` class launches a wizard to collect the user's Google API license key. In this case, this action is located in the upper-right corner of the search view.

Find the source code for the `LicenseKeyAction` class below:

Listing 13. LicenseKeyAction class

```
package com.ibm.developerworks.google.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.ui.IViewActionDelegate;
import org.eclipse.ui.IViewPart;

import com.ibm.developerworks.google.views.SearchView;
import com.ibm.developerworks.google.wizards.LicenseKeyWizard;

public class LicenseKeyAction implements IViewActionDelegate
{
    private SearchView searchView;

    public void init(IViewPart view)
    {
        this.searchView = (SearchView) view;
    }

    public void run(IAction action)
    {
        LicenseKeyWizard wizard = new LicenseKeyWizard();
        WizardDialog dialog = new WizardDialog(searchView.getViewSite()
                .getShell(), wizard);
        dialog.open();

    }

    public void selectionChanged(IAction action, ISelection selection)
    {

    }

}
```

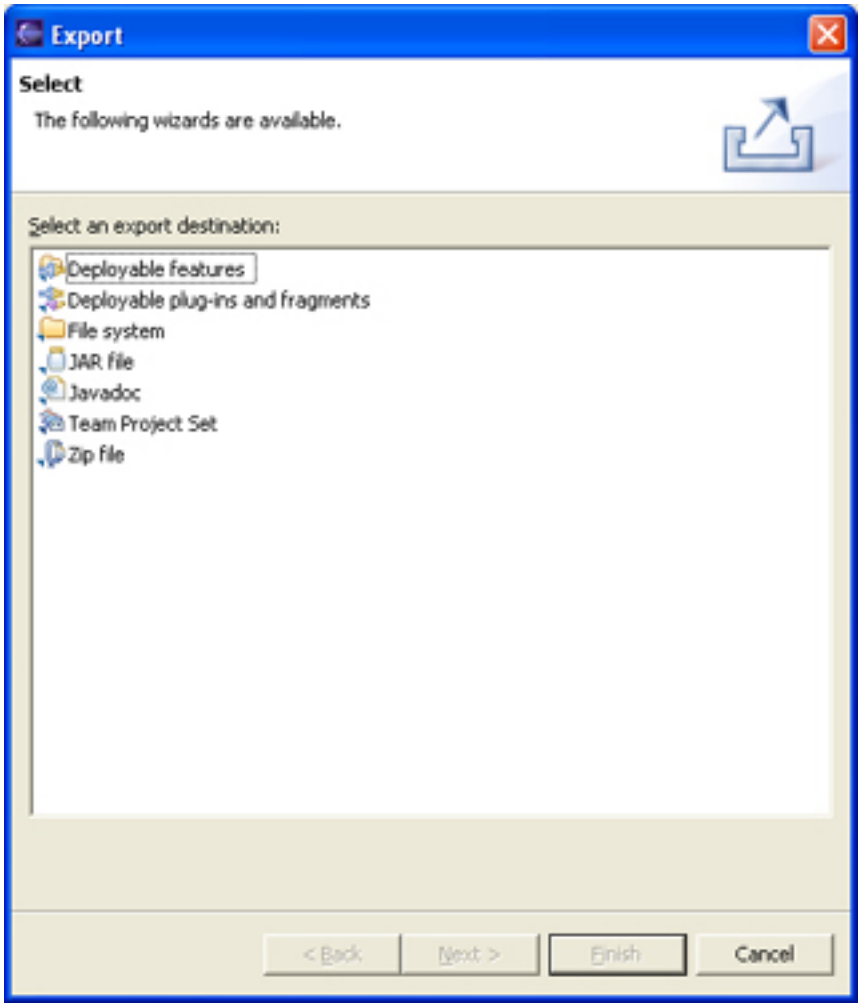
Before you run the Google application, verify that the project builds successfully and that you've received a license key to use Google's API.

Section 6. Launching the application

Exporting the application

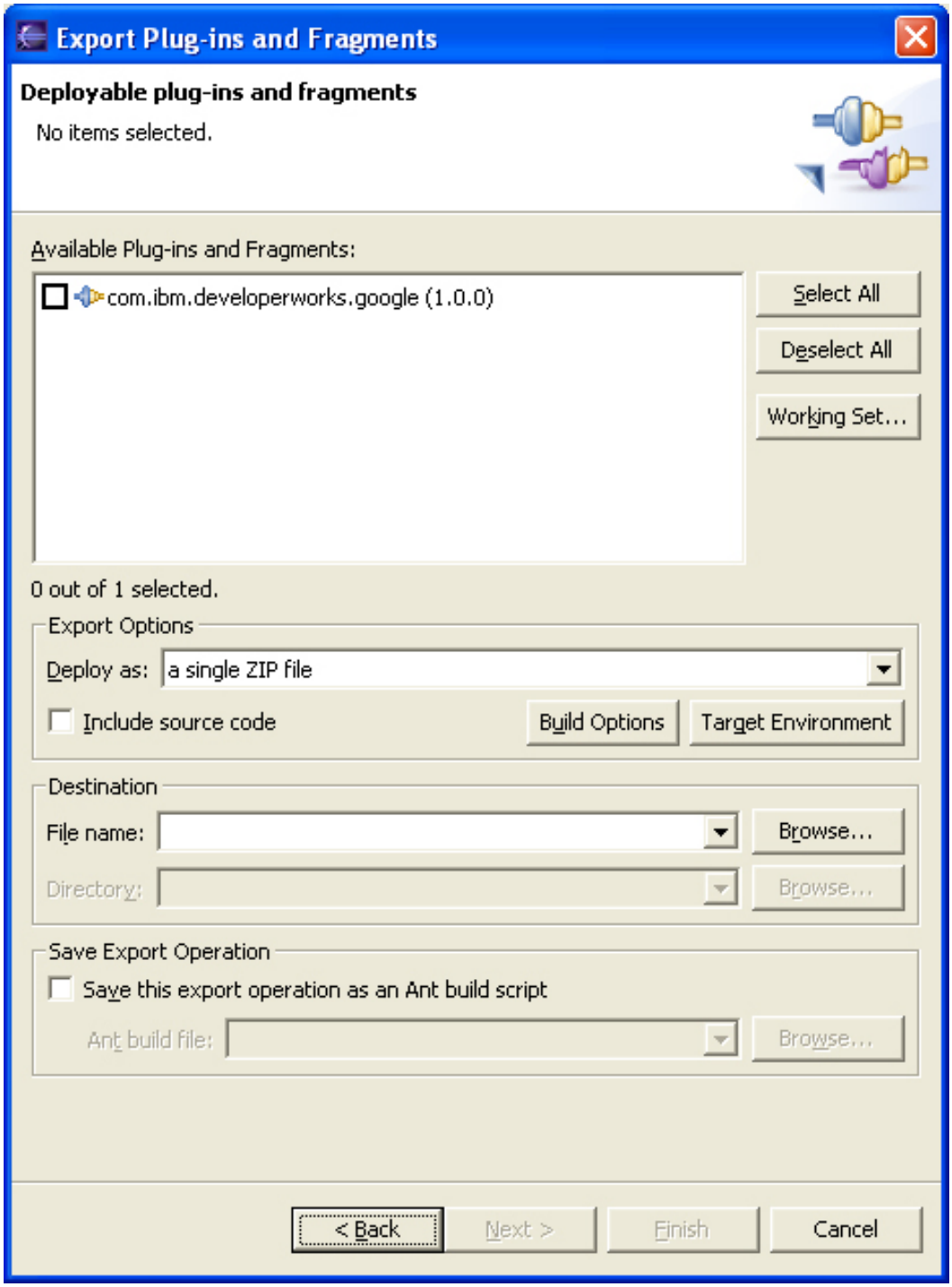
To create a stand-alone version of the Google application, complete the following steps within the Plug-in Development Environment:

1. Select **File > Export** from the menu bar to display the Export dialog.
   Figure 7. The Export dialog



2. Select **Deployable plug-ins and fragments** from the list of export options.

3. Click **Next** to display the Export Plug-ins and Fragments page of the Export wizard.
   Figure 8. Export Plug-ins and Fragments page of the Export wizard



4. Verify that the **com.ibm.developerworks.google** plug-in is checked.
5. Select **a directory structure** under the Deploy as field.
6. Click **Browse** and choose an export location.
7. Click **Finish** to build the project.

Preparing the directory structure

To complete the stand-alone application, you need to copy some files from the Eclipse IDE directory into Google's export directory. Unfortunately, Eclipse 3.0 doesn't provide a tool to copy all the necessary dependent plug-ins and JAR files into the export directory.

Complete the following steps to prepare the directory structure:

1. Copy startup.jar from the root directory of the Eclipse V3.0 IDE to the root of the Google application's export directory.
2. Copy the following directories from the Eclipse V3.0 IDE plug-in directory to the plug-in directory of the Google application's export directory:
   - org.eclipse.core.expressions_3.0.0
   - org.eclipse.core.runtime_3.0.0
   - org.eclipse.help_3.0.0
   - org.eclipse.jface_3.0.0
   - org.eclipse.osgi.services_3.0.0
   - org.eclipse.osgi.util_3.0.0
   - org.eclipse.osgi_3.0.0
   - org.eclipse.swt.win32_3.0.0 (Windows only)
   - org.eclipse.swt.gtk_3.0.0 (Linux only)
   - org.eclipse.swt_3.0.0
   - org.eclipse.ui.workbench_3.0.0
   - org.eclipse.ui_3.0.0
   - org.eclipse.update.configurator_3.0.0

Testing and executing the application

After you've completed the task of preparing the directory, your export directory should have the following structure:

Listing 14. Export directory structure

```
- google.bat (Windows only)
- google.sh (Linux only)
- startup.jar
+ ----- plugins
        + ----- org.eclipse.core.expressions_3.0
```

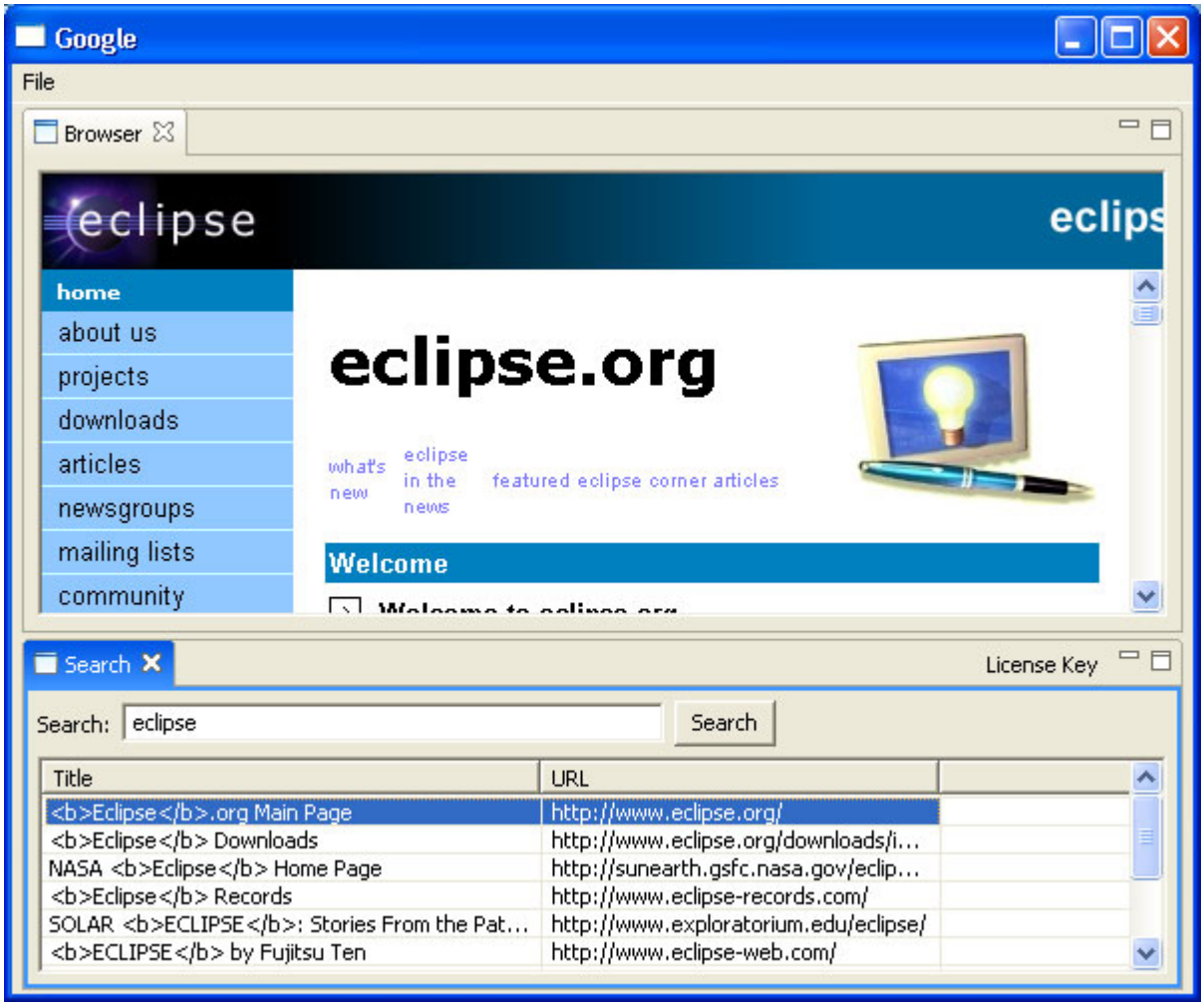
```
        + ----- org.eclipse.core.runtime_3.0.0
        + ----- org.eclipse.help_3.0.0
        + ----- org.eclipse.jface_3.0.0
        + ----- org.eclipse.osgi.services_3.0.0
        + ----- org.eclipse.osgi.util_3.0.0
        + ----- org.eclipse.osgi_3.0.0
        + ----- org.eclipse.swt.win32_3.0.0 (Windows only)
        + ----- org.eclipse.swt.gtk_3.0.0 (Linux only)
        + ----- org.eclipse.swt_3.0.0
        + ----- org.eclipse.ui.workbench_3.0.0
        + ----- org.eclipse.ui_3.0.0
        + ----- org.eclipse.update.configurator_3.0.0
```

To test the application, you need to create a launch script. Using your favorite text editor, create a file named google.bat (Windows®) or google.sh (Linux®) with the following content:

```
java -cp startup.jar org.eclipse.core.launcher.Main
-application com.ibm.developerworks.google.GoogleApplication
```

With all the classes created, the plug-in manifest defined, and all of the necessary dependencies in place, you can launch the Google application and perform a search. Figure 9 illustrates how you can use the Google API to search for the term "eclipse" and how the Eclipse project Web site is displayed.

Figure 9. Google RCP application with search results



Section 7. Summary

As the Eclipse development team begins to establish the RCP within the development landscape, it's going to be exciting to see how the platform's strategy and technology evolves. Although the RCP concept is very new, the 3.0 release of Eclipse delivers developers a framework they can start using today. The example Google application used in this tutorial demonstrates the generic workbench and explores how you can integrate various user-interface elements to create an elegant, cross-platform solution.

This series presented the following topics:

- An introduction to the core components that can make up an RCP application including Perspectives, Views, Actions, and Wizards.
- An exploration of how to develop an RCP through the use of extensions.

- A sample RCP application that you can use to query and display search results from Google.

Resources

Learn

- Read Understanding Layouts in SWT at Eclipse.org.
- Read Building and delivering a table editor with SWT/JFace at Eclipse.org.
- Read "Integrate ActiveX controls into SWT applications" (developerWorks, June 2003).
- "Developing JFace wizards" (developerWorks, May 2003) provides background.
- Read "Developing Eclipse plug-ins" (developerWorks, December 2002) to learn how to create Eclipse plug-ins using the Plug-in Development Environment's code-generation wizard.
- Read "XML development with the Eclipse Platform" (developerWorks, April 2003) for an overview of how the Eclipse Platform supports XML development.
- Browse all of the Eclipse content on developerWorks.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Stay current with developerWorks technical events and webcasts.

Get products and technologies

- Download Eclipse V3.0.
- Download Java 2 SDK, Standard Edition V1.4.2 from Sun Microsystems.
- Download Ant V1.6.1 or higher from the Apache Software Foundation.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Jeff Gunther
Jeff Gunther is the General Manager and founder of Intalgent Technologies, an emerging provider of software products and solutions utilizing the Lotus Notes/Domino and Java 2 Enterprise Edition platforms. Jeff Gunther has been a part of the Internet industry since its early, "pre-Mosaic" days. He has professional experience in all aspects of the software life cycle including specific software development expertise with Lotus Notes/Domino, Java/J2EE technology, DHTML,

XML/XSLT, database design, and handheld devices. You can contact him at jeff.gunther@intalgent.com.